Typische Bugs in nebenläufigen Java-Anwendungen automatisiert erkennen und testen

Christine Emrich

19. Januar 2013

Zusammenfassung

Fehler in nebenläufigen Programmen sind wegen des Nicht-Determinismus bei deren Ausführung oft schwer zu reproduzieren und stellen daher besonders hohe Anforderungen an das Testen. Diese Fehler folgen oft bekannten Fehlermustern und entstehen durch nicht beabsichtigte Kontextwechsel in ungesperrten kritischen Abschnitten. Sie können auf zwei unterschiedliche Arten erkannt werden: durch die Analyse des Quellcodes auf Fehlermuster oder durch das Ausführen des Programms mit möglichst vielen oder sehr gezielten Kontextwechseln. Kombinationen von existierenden Werkzeugen für statische und dynamische Code-Analyse können helfen, viele der Fehler zum Großteil automatisiert und früh im Entwicklungsprozess aufzudecken.

Schlüsselwörter: Testing, nebenläufige Programme, Fehlermuster, Fehler, Bugs, statische Code-Analyse, dynamische Code-Analyse

1 Einleitung

Mit der schnellen Verbreitung von Mehrkern-Prozessoren kommt heute kaum eine rechenintensive Anwendung mehr ohne Nebenläufigkeit aus. Nebenläufige Programme können typische Fehler enthalten, die aus der Entwicklung korrekter sequenzieller Programme unbekannt sind. Diese Fehler sind oft aufwändig zu reproduzieren und erfordern daher besondere Aufmerksamkeit beim (automatischen) Testen. Bugs in nebenläufigen Programmen entstehen hauptsächlich durch das lineare Denken von Entwicklern. Da auch nebenläufige Programme einem ähnliches Design und ähnlichen Mustern wie sequenzielle Programme folgen [7], wird der Nichtdeterminismus in der Ausführung oft übersehen.

Die so eingeführten Fehler stellen besondere Herausforderungen an das Testen: Sie zeigen oft deutlich andere Auswirkungen, als ihre sequenziellen Gegenstücke. Durch den Nichtdeterminismus in der nebenläufigen Ausführung treten sie oft nur selten und unter sehr speziellen Bedingungen auf. Diese Bedingungen sind meist nur unter großem Aufwand reproduzierbar [15] – die Gefahr, dass einmal beobachtete Bugs gar nicht erst erfasst werden

oder nicht bestätigt werden können (diese Art von Bugs werden gerne scherzhaft als „Loch-Ness-Monster-Bug“ oder „Bugfoot“ bezeichnet), ist hoch [11].

Ein weiteres Problem ist die Unsicherheit vieler Entwickler darüber, wie sie nebenläufige Programme effektiv testen, analysieren und debuggen können. Trotz umfangreicher Forschung zu diesem Thema, finden viele Methoden noch keine Anwendung im Programmieralltag. Stattdessen greifen viele Programmierer auf einfache aber unzuverlässige Last- und Stresstests zurück [14, 13].

Zudem sind verbreite Java-Testframeworks wie zum Beispiel JUnit häufig nicht auf das Testen von nebenläufigen Programmen ausgelegt. So ist es oft schwierig herauszufinden, in welchem Thread ein Fehler innerhalb eines Tests aufgetreten ist [9]. Viele der oben genannten Schwierigkeiten lassen sich allerdings mithilfe automatisierter Testwerkzeuge und der Kenntnis über die häufigsten Fehlerursachen einfach umgehen.

Abschnitt 2 gibt einen Überblick über gängige Ursachen von Fehlern in nebenläufigen Programmen. Abschnitt 2.1 erklärt das Konzept von kritischen Abschnitten und des kritischen Wettlaufs, auf den fast alle der Fehler zurückzuführen sind. In Abschnitt 2.2 werden Fehlermuster aufgeführt, in denen kritische Abschnitte fälschlicherweise für nicht unterbrechbar gehalten werden, während Abschnitt 2.3 Fehlermuster behandelt, die auf eine falsche Ausführungsreihenfolge von Befehlen zurückzuführen sind. In Abschnitt 2.4 wird ein Überblick über Fehlermuster gegeben, die durch den falschen Einsatz von Sperrmechanismen entstehen. Anschließend wird in Abschnitt 3 erklärt, das nebenläufige Programme entweder durch die Analyse des Codes auf typische Fehlermuster überprüft werden oder Bugs durch Ausführen des Programms provoziert werden können. Außerdem wird eine Übersicht über häufig genutzte Werkzeuge für statische und dynamische Code-Analyse gegeben. In Abschnitt 4 wird ein positives Fazit gezogen.

2 Fehlerursachen und -vermeidung

Im Folgenden sollen häufige Ursachen für Fehler in nebenläufigen Programmen betrachtet werden. Dafür ist es nötig, die Begriffe *kritischer Wettlauf* und *kritischer Bereich* zu erläutern und Vermeidungsstrategien für diese Probleme in Form von Sperren zu aufzuzeigen. Anschließend werden die drei wichtigen Fehlerkategorien *nicht-atomare Code-Segmente*, *Ausführungsreihenfolge* und *Verklemmung* anhand ihrer Ursachen im Denken von Programmierern voneinander abgegrenzt und häufige Fehlermuster beispielhaft angeführt. Eine Übersicht über die vorgestellten Fehlermuster bietet Tabelle 1.

2.1 Kritischer Wettlauf

Bugs sind Fehler, die während der Implementierung in ein Programm eingeführt werden [5]. In nebenläufigen Programmen entstehen sie vor allem durch *kritischen Wettlauf* (im Englischen *race condition*) [8]. Man spricht von kritischem Wettlauf, wenn das korrekte Ergebnis

Kategorie	Name
Nicht-atomare Code-Segmente	Nicht-atomare Java-Befehle
	Multi-Stage-Access
	Check-Then-Act
	Inkonsistente Synchronisierung
Ausführungsreihenfolge	sleep()-Fehlermuster
	Verlorener notify()-Befehl
Verklemmung	notify() statt notifyAll()
	Blockierender geschützter Bereich

Tabelle 1: Übersicht über gängige Fehlerkategorien und -muster

einer Operation von der Ausführungsreihenfolge ihrer Befehle abhängt. Falsche Ergebnisse oder inkonsistente Zustände können also durch „falsches Timing“ eines Kontextwechsels und eine dadurch ungünstige Ausführungsreihenfolge entstehen.

Kritische Abschnitte sind einzelne oder mehrere Befehle, die nicht durch andere Threads unterbrochen werden dürfen, die auf die gleichen Betriebsmittel zugreifen, da sonst kritischer Wettlauf entsteht. Um kritischen Wettlauf zu vermeiden, müssen diese Abschnitte im Code vom Programmierer explizit gesperrt werden. Sperren sorgen dafür, dass die Ausführung des gesperrten Bereichs nicht mehr durch andere Threads unterbrochen werden kann, die auf die gleichen Betriebsmittel zugreifen wie der geschützte Bereich.

Es gibt Befehle, die sich auch ohne Sperren nicht von anderen Threads unterbrechen lassen. Solche Operationen nennt man *atomar*, da sie immer als einzelner, unteilbarer Maschinenbefehl ausgeführt werden.

Die oben definierten Begriffe sollen an folgendem Beispiel verdeutlicht werden: Angenommen zwei Threads greifen auf einen gemeinsamen Zähler `s` zu und erhöhen dessen Wert gleichzeitig. `s` ist also das Betriebsmittel auf das beide Threads gleichzeitig zugreifen wollen. In Java würde der Befehl `s++` verwendet. Obwohl der Befehl wie eine atomare Operation aussieht, wird er tatsächlich in drei Schritten ausgeführt [5]: Zuerst wird der Wert von `s` gelesen, dann wird der gelesene Wert inkrementiert und der inkrementierte Wert anschließend in `s` zurückgeschrieben. Dies führt wie in Abbildung 1 zu sehen zu einer Reihe von möglichen Ausführungsreihenfolgen. Von diesen liefern diejenigen ein korrektes Ergebnis, in denen die drei Schritte des Inkrementieren-Befehls nicht unterbrochen werden. Diese Reihenfolge entspricht auch der Annahme des Programmierers. Folglich handelt es sich bei dem Befehl `s++` um einen kritischen Abschnitt. Da es auch – je nach Timing – Ausführungsreihenfolgen gibt, die zu einem falschen Ergebnis führen, handelt es sich bei diesem Beispiel um einen kritischen Wettlauf.

```
1 synchronized(sLock) {
2         s++;
3 }
```

Listing 1: Mit dem `synchronized`-Befehl umgesetzte Sperre.

Um den kritischen Wettlauf in obigem Beispiel zu vermeiden, muss der kritische Abschnitt mit einem gemeinsamen Sperrobjekt (`sLock`) gesperrt werden. Dies kann wie in Listing 1 zu sehen mit dem Java-eigenen `synchronized`-Befehl geschehen.

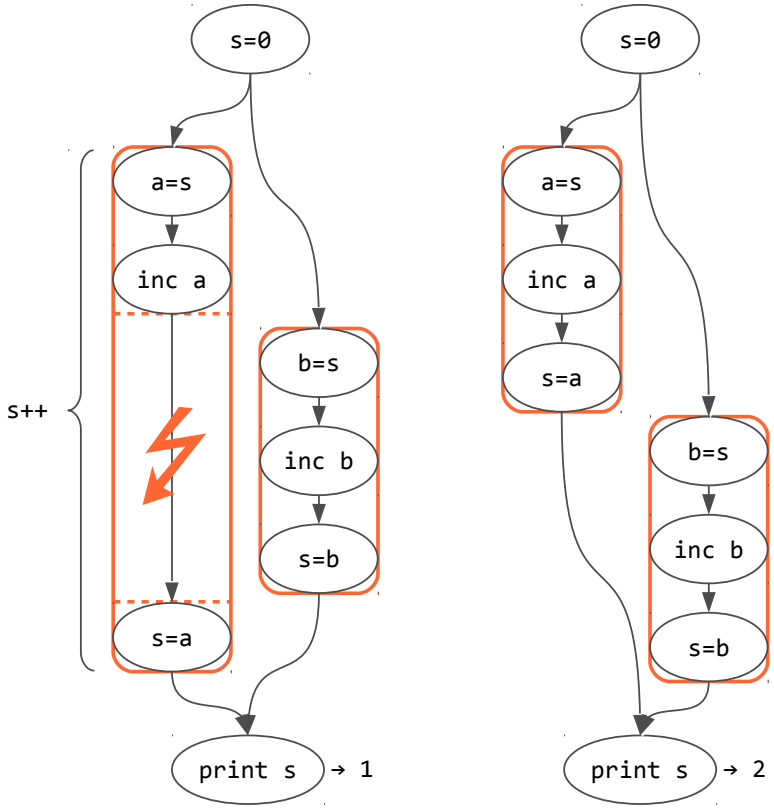

Abbildung 1: Zwei mögliche Ausführungsreihenfolgen mit unterschiedlichem Ergebnis: Die linke führt zu einem falschen Ergebnis, weil die Operation innerhalb des kritischen Abschnitts unterbrochen wird. Die rechte führt zu einem korrekten Ergebnis.

2.2 Nicht-atomare Code-Segmente

Viele Bugs entstehen dadurch, dass der Programmierer ein Code-Segment fälschlicherweise für nicht von anderen Threads unterbrechbar hält. In einer Studie von Lu et al. [11] wurde beobachtet, dass fast die Hälfte der dort untersuchten Bugs in diese Kategorie fallen.

Nicht-atomare Java-Befehle Eine häufig gemachter Fehler in diesem Bereich ist die Annahme, dass ein einzelner Befehl einer Programmiersprache grundsätzlich atomar sein muss (vgl. Beispiel in Abschnitt 2.1 und Abbildung 1). Diese Annahme führt dazu, dass betroffene Befehle nicht explizit geschützt werden und durch nebenläufige Ausführung nicht korrekte Ergebnisse entstehen. Diese Ergebnisse setzten sich im weiteren Programmfluss fort und können zu inkonsistenten Zuständen führen [7, 5].

Multi-Stage-Access Manchmal reicht es nicht aus, einzelne Operationen vor Unterbrechung durch andere Threads zu schützen. Das ist zum Beispiel der Fall, wenn zwei für sich genommen threadsichere Objekte voneinander abhängen. Um inkonsistente Zustände zu vermeiden, müssen Zugriffe, die beide Variablen betreffen, als ganzes geschützt werden.

Ein Beispiel findet sich in Listing 2: Obwohl die Variablen `account` und `bookingCount` für sich genommen beide geschützt sind, handelt es sich bei den Zeilen sieben bis acht um einen nicht gesperrten kritischen Abschnitt (siehe auch Abbildung 2). Findet zwischen diesen Zeilen ein Kontextwechsel statt, so sieht der unterbrechende Thread einen inkonsistenten Zustand: Obwohl eine Buchung stattgefunden hat, wurde diese in `bookingCount` noch nicht berücksichtigt. Die Berechnung der durchschnittlichen Buchungshöhe würde in diesem Fall zum Beispiel falsche Ergebnisse liefern. Bugs nach diesem Muster, nennt man *Multi-Stage-Access* [8, 5]. Sie können durch explizites Sperren des gesamten kritischen Abschnitts behoben werden.

```
1 AtomicLong final account; // Kontosumme
2 AtomicLong final bookingCount; // Anzahl der Buchungen auf das Konto
3
4 /* Kontosumme um Betrag erhoehen und
5 Anzahl der Buchungen um eins erhoehen */
6 public void addToAccount(long amount) {
7       account.addAndGet(amount);    // kritischer...
8       bookingCount.incrementAndGet(); // ...Abschnitt
9 }
```

Listing 2: Code-Beispiel für Multi-Stage-Access.

Check-Then-Act Einer der häufigsten Typen von kritischem Wettlauf nennt sich *Check-Then-Act*. Hier wird abhängig vom Ergebnis einer Abfrage weiterer Code ausgeführt. Dabei wird allerdings nicht beachtet, dass zwischen der Abfrage und den weiteren Operationen ein Kontextwechsel stattfinden kann. Diese Unterbrechung kann dazu führen, dass das Ergebnis der Abfrage nicht mehr valide ist [8]. Ein Beispiel findet sich in Abbildung 3: Dort findet zwischen der Abfrage, ob eine Datei bereits geöffnet ist und dem Schreibvorgang auf die Datei ein Kontextwechsel statt, der dafür sorgt, dass die Datei geschlossen wird. Dementsprechend schlägt der Schreibvorgang trotz voriger Überprüfung fehl.

Inkonsistente Synchronisierung[1] Um kritische Abschnitte vor ungewollten Kontextwechseln zu schützen, wird der Programmierer zum Sperren dieses Bereichs gezwungen. Alle kritischen Abschnitte mit Zugriff auf das gleiche Betriebsmittel müssen dabei auch in der gleichen Art und Weise gesperrt werden. Das führt schnell zu Leichtsinnsfehlern. Inkonsistente Synchronisierung entsteht dadurch, dass nicht alle Zugriffe auf eine gemeinsames

[1] im Englischen auch „wrong lock or no lock bug“

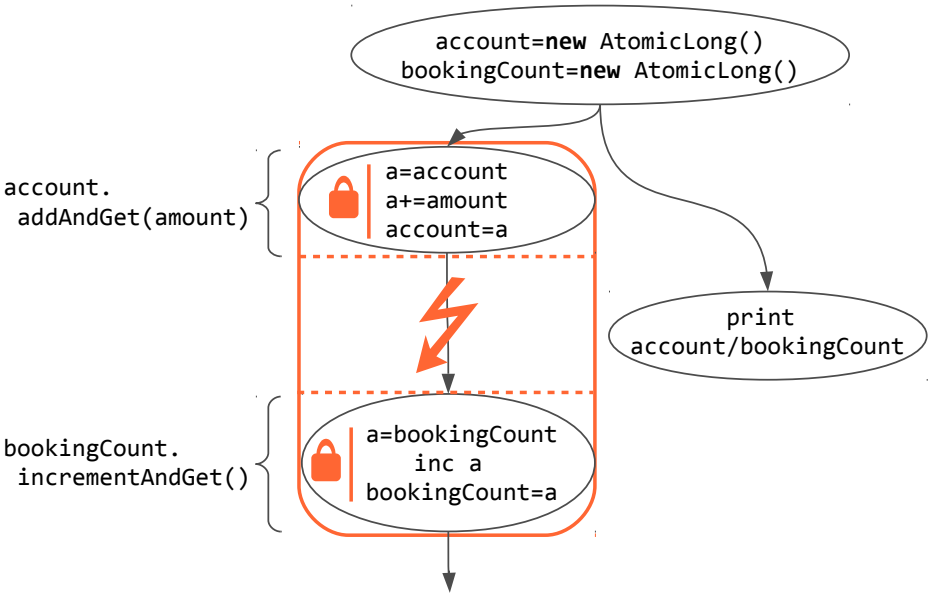

Abbildung 2: Typischer Multi-Stage-Access-Fehler: Der Zugriff auf die Variablen `account` und `bookingCount` ist zwar gesperrt, aber die Sperren decken nicht den gesamten kritischen Abschnitt ab. Dadurch kann die Berechnung der durchschnittlichen Buchungshöhe falsche Ergebnisse liefern.

Betriebsmittel (z.B. ein Objekt) geschützt werden. Das kann entweder geschehen, wenn ein Zugriff gar nicht gesperrt oder eine falsche Sperre benutzt wird (z.B. ein falsches Sperrobjekt) [2, 5]. Das vermeidlich geschützte Betriebsmittel ist dadurch nicht mehr ausreichend vor nebenläufigen Zugriffen geschützt, sodass es zu inkorrektem Verhalten und Ergebnissen kommen kann.

2.3 Ausführungsreihenfolge

In diese Kategorie fallen Bugs, die dadurch entstehen, dass der Programmierer eine bestimmte Ausführungsreihenfolge von nebenläufigen Befehlen annimmt, sie aber nicht durchsetzt. Laut einer Studie von Lu et al. [11] wird diese Gruppe von Bugs von der Forschung nicht ausreichend beachtet, obwohl sie durchaus in parallelen Programmen zu finden sind. Dabei kann diese Fehlergruppe sehr unterschiedliche Auswirkungen zeigen und sowohl die Korrektheit eines Programms als auch den Programmfortschritt gefährden.

`sleep()`-Fehlermuster Manchmal verwenden Programmierer statt des `join()`-Befehls `sleep()`, um darauf zu warten, dass ein Kindprozess abgearbeitet wurde. Das kann zu Fehlern führen, wenn der Kindprozess langsamer abgearbeitet wird, als vom Programmierer angenommen. Der aufrufende Prozess fährt in diesem Fall fort, obwohl Ergebnisse des Kindprozesses ggf. noch nicht vorhanden sind [5].

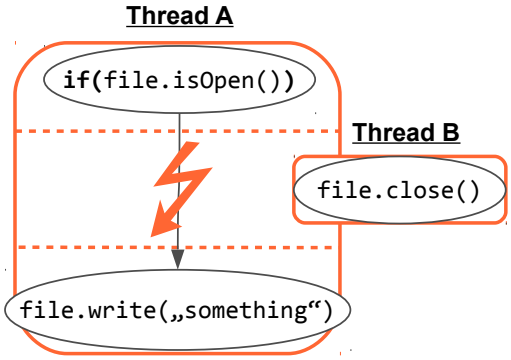

Abbildung 3: Beispiel für einen typischen *Check-Then-Act*-Fehler. Obwohl vorher geprüft wurde, ob die Datei geöffnet ist, schlägt der Schreibvorgang unten im Bild fehl.

Verlorener `notify()`-Befehl Wird ein `notify()`-Befehl ausgeführt, bevor der zugehörige `wait()`-Befehl aufgerufen wurde, so hat der `notify()`-Befehl keine Wirkung. Da der `notify()`-Befehl nun für den wartenden Thread „verloren“ ist, kann dieser blockieren [5]. Das Programm kann also stoppen, weil der Programmierer eine bestimmte Ausführungsreihenfolge der beiden Befehle annimmt, die unter Umständen nicht eintrifft.

2.4 Verklemmung

Um eine Sequenz von Befehlen ohne Unterbrechung ausführen zu können, bietet Java Sperrmechanismen wie den `synchronize`-Block oder Semaphoren. Diese Mechanismen können helfen, die Korrektheit eines Programms trotz seiner nebenläufigen Ausführung zu wahren [9]. Alle Bugs aus Sektion 2.2 und fast alle aus Sektion 2.3 können damit behoben werden. Allerdings werden durch die Sperrmechanismen eine neue Klasse von Bugs in parallele Programme eingeführt: *Verklemmungen*. Diese Bugs haben die Auswirkung, dass der Programmfluss stoppt. In sequenziellen Programmen sind diese Art von Bugs – bis auf Endlosschleifen – weitgehend unbekannt [7].

Die häufigste Ausprägung dieser Fehlerklasse ist der *Deadlock*. Während eines Deadlocks warten ein oder mehrere Threads auf Ressourcen, die jeweils von ihm selbst oder einem anderen Thread gehalten werden [9]. Dadurch entsteht eine Situation, in der keiner der beteiligten Threads mit seiner Aufgabe fortfahren kann und das Programm „hängt“. Im Gegensatz zu Datenbanksystemen kann sich die JVM nicht selbst von Deadlocks erholen.

Es ist zu beachten, dass Deadlocks auch durch nur einen einzigen Thread entstehen können, der auf Ressourcen wartet, die von ihm selbst gehalten werden[2]. Außerdem sind an fast allen Deadlock-Bugs höchstens zwei Ressourcen und höchstens zwei Threads beteiligt [11].

[2] Das gilt nicht für Sperren mithilfe des `synchronized`-Befehls, da dieser nur Zugriffe von *anderen* Threads ausschließt.

`notify()` statt `notifyAll()` Der Befehl `notifyAll()` benachrichtigt alle auf die entsprechende Ressource wartenden Threads und weckt sie so auf. Der Befehl `notify()` hingegen benachrichtigt genau einen der wartenden Threads. In manchen Fällen führt der unbeabsichtigte Aufruf des Befehls `notify()` anstatt von `notifyAll()` dazu, dass wartende Threads nicht oder zu spät aufgeweckt werden [2].

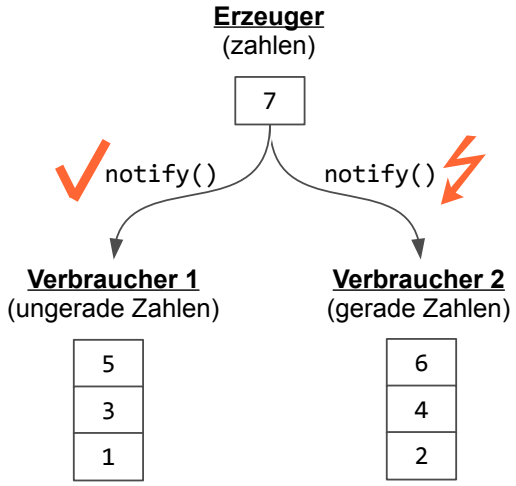

Abbildung 4: Ein einfaches Erzeuger-Verbraucher-Szenario, in dem ein `notify()`-Befehl zur Verklemmung führen kann.

Abbildung 4 zeigt ein einfaches Erzeuger-Verbraucher-Szenario in dem der Fehler auftreten kann: Der Erzeuger liefert Zahlen, während ein Verbraucher gerade und der andere ungerade Zahlen verarbeiten kann. Wenn nun beide Verbraucher schlafen und der Erzeuger eine ungerade Zahl zur Verfügung stellen möchte, kann der Aufruf von `notify()` durch den Nichtdeterminismus der JVM zu zwei unterschiedlichen Ergebnissen führen. Wenn der Verbraucher der ungeraden Zahlen aufgeweckt wird, kann dieser die Zahl verarbeiten. Wird aber der andere Verbraucher aufgeweckt, kann er die ungerade Zahl nicht verarbeiten und legt sich schlafen. Wenn der Erzeuger nun – zum Beispiel wegen eines vollen Puffers – nicht weiterarbeitet, erfolgen keine weiteren `notify()`-Aufrufe, wodurch eine typische Verklemmung entsteht. Diese Verklemmung kann durch das Ersetzten von `notify()` mit `notifyAll()` vermieden werden.

Blockierender geschützter Bereich Wenn ein geschützter Bereich seine Ressourcen (z.B. reservierte `Lock`Objekte) nicht mehr abgibt, kann es im schlimmsten Fall zu Deadlocks kommen. Wie in Listing 3 zu sehen, tritt dieser Fehler häufig auf, wenn `Lock`-Objekte durch eine auftretende Exception nicht freigegeben werden [2]. Auch blockierende I/O-Operationen innerhalb eines geschützten Bereichs können zu diesem Fehler führen [5].

Dieser Fehler kann vermieden werden, indem potenzielle Exceptions mit einem `try-catch`-Block umschlossen werden. Anschließend sollte die Sperre in einem angeschlos-

senen `finally`-Block wieder aufgehoben werden, da dieser auch im Fall einer Ausnahme ausgeführt wird.

```
1 Lock.acquire();
2 // kann eine Exception werfen und Codeausfuehrung hier abbrechen:
3 doSomething();
4 //dieser Befehl wird nach einer Exception nicht ausgefuehrt:
5 Lock.release();
```

Listing 3: Code-Beispiel für einen potentiell blockierenden geschützten Bereich

3 Erkennungs- und Testmethoden

Wie bereits gesehen, können Bugs in nebenläufigen Programmen sowohl nach Ursache als auch nach ihrer Auswirkung auf Korrektheit und Programmfortschritt unterschieden werden. Für jede dieser Kategorien gibt es geeignete und unzureichende Erkennungs- und Testmethoden, die im Folgenden genauer beschrieben werden. Prinzipiell lässt sich zwischen den zwei großen Gruppen der dynamischen und der statischen Analyseverfahren unterteilen.

Unabhängig von dem gewählten Verfahren muss zuerst die sequenzielle Korrektheit des Programms sichergestellt werden. Erst wenn wenn keine Fehler mehr auftreten, die nicht auf die nebenläufige Ausführung eines Programms zurückzuführen sind, sollte man sich auf kompliziertere parallele Analyseverfahren konzentrieren [9].

3.1 Statische Code-Analyse

Unter statischer Code-Analyse versteht man das Analysieren von Quellcode, ohne ihn ausführen zu müssen [9]. Dies geschieht meist mithilfe von automatisierten Werkzeugen, die nach formalen Fehlern oder typischen Fehlermustern suchen. Das macht die statische Code-Analyse zu einem einfachen und günstigen Analyseverfahren. Eine Übersicht über bekannte Werkzeuge für statische Analyse in Java findet sich in Tabelle 2.

Name	**Beschreibung**
Findbugs [18]	Open-Source-Tool für statische Code-Analyse von Java-Bytecode
Jlint [20]	Open-Source-Tool um Bugs, Inkonsistenzen und Synchronisationsprobleme (unter anderem durch Flusskontrolle) zu finden
Jtest [21]	unter anderem statische Code-, Fluss-Statistik- und Metriken-Analyse

Tabelle 2: Auswahl bekannter statischer Code-Analyse-Tools

Statische Analyse-Werkzeuge finden Fehler durch ihre Ursache im Quellcode, nicht aber durch ihre Auswirkungen. Das bringt den Vorteil mit sich, dass oft gemachte, gängige Fehler gut automatisiert erkannt werden können. Dies gelingt allerdings nur, wenn die Fehler im Code gut zu identifizieren sind (siehe Abbildung 5). Fehler mit komplizierteren Ursachen, wie z.B. Deadlock-Bugs, können mithilfe der statischen Code-Analyse nur schwer gefunden werden [10].

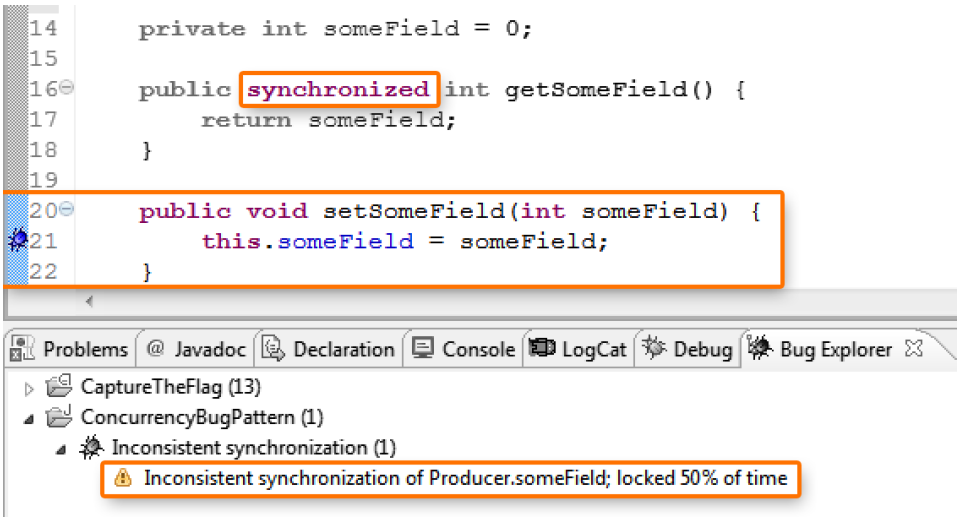

Abbildung 5: FindBugs [18] findet einfache inkonsistente Synchronisierungen zuverlässig, hebt die Fehlerstelle hervor und nennt das betroffene Objekt.

Ein weiteres Problem ist, dass die Anzahl der von statischen Analyse-Tools falsch erkannten Fehler häufig sehr hoch ist [9, 12, 3]. Diese vermeintlichen Fehler müssen von den Entwicklern von Hand gegengeprüft und bestätigt werden [1]. Zudem kann die statische Code-Analyse verschiedenen Programmeingaben nicht berücksichtigen [3].

In einer Studie von Mamun et al. [12] wurde herausgefunden, dass sowohl kommerzielle als auch freie statische Analyse-Werkzeuge insgesamt viele Fehlermuster in nebenläufigen Programmen unterstützen. Allerdings entdeckten sie mit durchschnittlich 25% nur sehr wenige der untersuchten Fehler. Außerdem unterscheiden sich verschiedene Werkzeuge stark darin, welche Fehlertypen sie gut erkennen, sodass ein Werkzeug allein im Verhältnis nur wenige Fehlermuster erkennen kann.

Werkzeuge zur statischen Analyse empfehlen sich also vor allem als günstige und praktische Ergänzung zu anderen Analyseverfahren. Um eine möglichst große Bandbreite von Fehlern in nebenläufigen Programmen aufzudecken, ist es außerdem ratsam, mehr als ein statisches Analyse-Werkzeug zu verwenden [12, 10]. Zudem eignen sich diese Werkzeuge gut zur Vorbereitung und Unterstützung von Code-Reviews.

3.2 Dynamische Code-Analyse

In der dynamischen Code-Analyse wird ein Programm auf Fehler überprüft, indem es ausgeführt wird. Viele Bugs in nebenläufigen Programmen werden durch ungewollte Kontextwechsel innerhalb von kritischen Abschnitten verursacht. Die dynamische Code-Analyse hat deswegen zum Ziel, möglichst viele Kontextwechsel zu provozieren, um Unterbrechungen in ungeschützten kritischen Abschnitten zu verursachen (siehe Abbildung 6). Im Gegensatz zur statischen Code-Analyse fallen Fehler in der dynamischen Code-Analyse also nicht durch ihre Ursache im Quellcode, sondern durch ihre Auswirkung auf. Inkorrekte Zustände und Ergebnisse oder ein „Hängen“ des Programms werden ähnlich wie bei sequenziellen Tests zuerst provoziert und dann versucht zu erkennen.

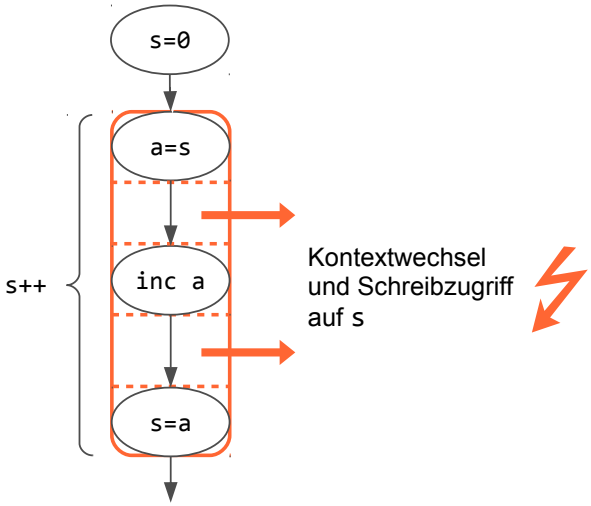

Abbildung 6: Schon bei einfachen Befehlen können Kontextwechsel in ungeschützten kritischen Abschnitten zu Fehlern führen. Die dynamische Code-Analyse versucht, genau diese Kontextwechsel zu provozieren.

Kontextwechsel innerhalb kritischer Abschnitte können durch verschiedene Methoden provoziert werden: Indem viel Last erzeugt wird, durch das Einführen von unterbrechenden Code oder durch das systematische Durchspielen aller möglichen Kontextwechsel. Auf die drei Ansätze wird im Folgenden genauer eingegangen. Eine Auswahl bekannter Werkzeuge für die dynamische Code-Analyse findet sich in Tabelle 3.

Eine große Gefahr bei der dynamischen Code-Analyse geht vom Testcode selbst aus. Testcode kann ein anderes Timing und andere Synchronisationseigenschaften in das getestete Programm einführen, als im Produktiveinsatz. Das kann dazu führen, dass im normalen Betrieb auftretende Bugs während des Tests schlicht nicht reproduziert werden können. In solchen Fällen spricht man von sogenannten *Heisenbugs* [9, 13]. Um solches Verhalten zu vermeiden, sollte innerhalb von Tests der Zugriff auf zu testende Programmeigenschaften möglichst nicht synchronisiert werden, da sonst Nebenläufigkeit künstlich ausgebremst wird [9]. Zudem sollten Tests auf unterschiedlichen Systemen mit verschiedenen Architek-

Name	Hersteller	Beschreibung
ConTest [17]	IBM	Bekanntes Testing-Tool, das die Wahrscheinlichkeit für das Auftreten von Synchronisationsproblemen erhöht. Außerdem zeigt es die Test- und Synchronisationsabdeckung des Programms an und bietet die Möglichkeit Teile der Programmausführung zu wiederholen.
Java Pathfinder [19]	NASA	Eine JVM, deren Kerngebiet das Model-Checking ist. Es kann lokale Variablen, Stack, Heap und den Status von Threads während der Laufzeit überwachen. Mögliche Ausführungsreihenfolgen werden durch State-Matching, Backtracking und Partial-Order-Reduction auf sinnvoll zu testende Ausführungsreihenfolgen reduziert.
CHESS [16]	Microsoft	Ein Model-Checker für C# und C++, der vom Benutzer angegebene „Szenarien“ testet. Teile der Programmausführung können deterministisch wiederholt werden.

Tabelle 3: Auswahl bekannter dynamischer Code-Analyse-Tools

turen oder Betriebssystemen ausgeführt werden, um Fehler aufzudecken, die nicht auf allen Systemen auftreten.

Um ein Programm durch dynamische Code-Analyse auf seine Thread-Sicherheit überprüfen zu können, müssen Tests mit mehreren Threads auf das Programm zugreifen. Dabei ist es sinnvoll, die Tests auf Mehrkernprozessoren durchzuführen. Außerdem sollten mehr Threads als CPUs eingesetzt werden. Beide Maßnahmen zusammen sorgen dafür, dass viele Threads vom Betriebssystem regelmäßig schlafen gelegt werden müssen und so mehr und schlechter vorhersehbare Kontextwechsel stattfinden [7].

Um noch mehr Kontextwechsel zu provozieren, sollten die Threads des Testprogramms außerdem gleichzeitig gestartet werden und lange genug laufen. Damit entsteht eine möglichst große Laufzeitüberschneidung und es wird weniger wahrscheinlich, dass die Threads (im schlimmsten Fall) sequenziell ausgeführt werden.

Weitere Ausführungsreihenfolgen können eventuell provoziert werden, indem `Thread.yield()`- oder `Thread.sleep()`-Befehle[3] in den Programmcode eingefügt werden. Sie fordern die JVM dazu auf, einen Kontextwechsel durchzuführen. Das macht besonders zwischen Operationen Sinn, die auf gemeinsame Variablen zugreifen. Um diese Befehle nicht von Hand in bestehenden Code einfügen und wieder entfernen zu müssen, gibt es automatisierte Werkzeuge, die dem Programmierer diese Aufgabe abnehmen [9].

Eine besondere Schwierigkeit bei der dynamischen Code-Analyse ist das Testen von Programmfortschritt [6]. Es ist oft schwierig vorauszusagen, wie viel Zeit eine bestimmte Operation zum Ausführen benötigt und ab wann ein Test fehlschlagen muss. Zudem sind blockierende Methoden oft sogar erwünscht, z.B. beim typischen *Consumer-Producer-Pro-*

[3] Solche Befehle zu Testzwecken werden auch als „noise“ bezeichnet [15].

blem. Dieses Verhalten muss ebenfalls explizit getestet werden. Dabei muss der betreffende Test fehlschlagen, wenn der Thread ohne zu blockieren durchlaufen wird [9]. Der benötigte Testcode ist aufwändig und aufgrund der nur geschätzten Ausführungszeiten oft selbst fehleranfällig.

Sollen während der Analyse alle möglichen Kontextwechsel durchgespielt werden, dann müssen neben verschiedenen Eingaben auch alle möglichen Ausführungsreihenfolgen des Programms getestet werden, um tatsächlich alle Fehler provozieren zu können. Dies ist in der Praxis selbst für kleine Programme problematisch, da die möglichen Ausführungsreihenfolgen exponentiell zur Anzahl der Threads steigen [11, 13, 1]. Es bleiben zwei Vorgehensweisen, um trotz der Beschränkung möglichst viele Fehler provozieren zu können: Entweder wird versucht so viele Ausführungsreihenfolgen wie möglich zufällig abzudecken [9, 3] oder Kontextwechsel werden systematisch nur an den „richtigen“, besonders fehleranfälligen Stellen durchgeführt.

Ein systematischer Ansatz der dynamischen Code-Analyse stellt das *Model-Checking* dar. Hier werden alle möglichen Kontextwechsel systematisch durchgespielt. Dieser Ansatz benötigt allerdings einen deterministischen und kontrollierbaren Scheduler [13]. Außerdem ist er durch die hohe Anzahl an möglichen Ausführungsreihenfolgen sehr rechenintensiv und skaliert schlecht [3]. Andererseits gleicht der systematische Ansatz einen großen Nachteil von nichtdeterministischen Tests aus: Er kann nicht nur Fehler provozieren und so deren Existenz nachweisen, sondern ebenfalls die Bedingungen (z.B. die Ausführungsreihenfolge) aufzeigen, unter denen der Fehler auftritt [15]. Diese Eigenschaft ist von großem Vorteil, um gefundene Fehler verstehen und beheben zu können.

4 Fazit

Fehler in nebenläufigen Programmen haben ihre Ursache vor allem im sequenziellen Denken von Programmierern und dem daraus resultierenden fehlerhaften Umgang mit kritischen Abschnitten und Sperrmethoden. Das Erkennen dieser Fehler durch statische und dynamische Code-Analyse unterscheiden sich deutlich in Performance, Komplexität und ihrer Fähigkeit Ursachen oder Auswirkungen dieser Fehler zu finden. Allerdings gibt es für beide Methoden bewährte Werkzeuge, die Entwicklern viel Aufwand abnehmen. Als Programmierer möchte man sowohl möglichst viele Fehler in nebenläufigen Programmen finden als auch möglichst viele Informationen zu deren Ursachen erhalten. Daher empfiehlt es sich, schon früh im Entwicklungsprozess eine Mischung aus mehren Werkzeugen einzusetzen [4, 3]. Für die Zukunft wären Testwerkzeuge wünschenswert, die beide besprochenen Methoden in sich vereinen, um von deren verschiedenen Vorteile zu profitierten und deren Nachteile auszugleichen. Die Kombination von statischer und dynamischer Code-Analyse ist ein aktives Forschungsgebiet [3].

Literaturverzeichnis

[1] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. *Software Engineering Conference, Australian*, 0:0068, 2001.

[2] Jeremy S. Bradbury and Kevin Jalbert. Defining a catalog of programming anti-patterns for concurrent java. In *Proc. of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, pages 6–11, Oct. 2009.

[3] Jun Chen and Steve MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, PADTAD '08, pages 8:1–8:9, New York, NY, USA, 2008. ACM.

[4] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.

[5] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *Parallel and Distributed Processing Symposium, International*, 0:286b, 2003.

[6] Jan Fiedor, Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. A uniform classification of common concurrency errors. In *Proceedings of the 13th international conference on Computer Aided Systems Theory - Volume Part I*, EUROCAST'11, pages 519–526, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency in Practice. pages 1–11. Addison-Wesley Professional, May 2006.

[8] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency in Practice. pages 15–32. Addison-Wesley Professional, May 2006.

[9] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency in Practice. pages 247–274. Addison-Wesley Professional, May 2006.

[10] Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. How Good is Static Analysis at Finding Concurrency Bugs? In *Source Code Analysis and Manipulation*, 2010.

[11] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, March 2008.

[12] M. A. A. Mamun, A. Khanam, H. Grahn, and R. Feldt. Comparing four static analysis tools for java concurrency bugs. In *Proc. of the Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 143–146, 2010.

[13] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. 1 chess: A systematic testing tool for concurrent software. 2010.

[14] Neha Rungta and Eric G. Mercer. Clash of the titans: tools and techniques for hunting bugs in concurrent programs. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 9:1–9:10, New York, NY, USA, 2009. ACM.

[15] Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 27–38, New York, NY, USA, 2007. ACM.

Online-Quellen

[16] Chess: Systematic concurrency testing. *https://chesstool.codeplex.com/*. [letzter Zugriff: Nov. 2012].

[17] Contest – a tool for testing multi-threaded java applications. *https://www.research.ibm.com/haifa/projects/verification/contest/*. [letzter Zugriff: Nov. 2012].

[18] Findbugs™ – find bugs in java programs. *http://findbugs.sourceforge.net/*. [letzter Zugriff: Nov. 2012].

[19] Java™ pathfinder ...the swissarmy knife of java™ verification. *http://babelfish.arc.nasa.gov/trac/jpf*. [letzter Zugriff: Nov. 2012].

[20] Jlint. *http://jlint.sourceforge.net/*. [letzter Zugriff: Nov. 2012].

[21] Parasoft. Parasoft® jtest®: Java testing, static analysis, code review. *http://www.parasoft.com/jsp/products/jtest.jsp*. [letzter Zugriff: Nov. 2012].